CHAPTER 9

Programming ISAPI Filters

In This Chapter

- Learning the theory behind ISAPI Filters
- Understanding the MFC classes used for ISAPI Filters
- Determining whether you need to create an ISAPI Extension or ISAPI Filter
- Creating a simple ISAPI Filter
- Defining test suites for ISAPI Filters
- Learning about the various types of ISAPI Filters
- Creating ISAPI Filters that post-preprocess Web data
- Creating an ISAPI Filter for client authentication
- Rerouting the user using an URL-mapping request ISAPI Filter
- Logging IIS data requests using an ISAPI Filter

Controlling the flow of data between the client and the server is what most of the programming for any Web server is all about. Data-flow control takes many forms including servicing client queries, maintaining server security, and querying the client for information. In addition to these common processing needs, are more esoteric needs like redirecting the client to an alternative Web page when needed, or providing something more specific than a general server error message when a problem occurs.

In Chapter 8, we looked at ISAPI Extensions, which allow you to control the flow of data using the same techniques that you might find in any script (with the addition of full server access). The main consideration for the programmer is that all of this processing takes place using the raw client request. In other words, using this form of processing means that you don't get a chance to work with the data during the various phases of server processing.

In this chapter, we're going to look at another form of data-flow control that's provided by an ISAPI Filter. Using an ISAPI Filter allows you to provide a finer level of control over the data flow between the client and server. ISAPI Filters are actually placed between the client and server during various phases of server-side processing. In effect, you get to see the data as it flows through the server and react to the data during that phase of processing. You can monitor and modify the data flow between the client and the server in seven different processing phases. We'll discuss all of the monitoring options you have in the first section of the chapter, "Understanding ISAPI Filters".

Once you understand the basic theory behind ISAPI Filters, we'll look at the essential question of when to use an ISAPI Filter or an ISAPI Extension. The second section of the chapter

will help you understand how each of these technologies fits into an overall Web-server strategy. (Chapter 8 partially answered these questions, so you may want to look at the theory sections of that chapter as well.) Most important of all, we'll discuss when using ISAPI of any kind is overkill. Often, using a script is faster and easier than using ISAPI. A good rule of thumb to remember is that ISAPI is for complex programming situations in which you require full control over the processing sequence. In other words, ISAPI is the answer for those situations in which the complexity of writing a complete script becomes overwhelming and the script wouldn't provide the level of control you needed.

Don't test any of the examples in this chapter on a production system or you may experience some type of damage to your system. None of the examples in this chapter are designed for production use. They're all designed to provide an overview of how you can use ISAPI Filters to create specific kinds of server aids. In other words, you should use these examples on a test server setup to learn how ISAPI can help you create a more flexible and higher-speed Web site. In addition, you'll most likely need to make some small changes to the ISAPI Filter code (like redirection URLs) to make the examples work on your system because your Web-server setup will likely differ from the one I used.

The third section of the chapter will help you write a very simple ISAPI Filter. The goal of this section is to teach you ISAPI Filter-writing basics, not to help you write a typical production ISAPI Filter. This basic example will provide you with an overview of ISAPI Filter programming techniques; you can skip it if you're already familiar with ISAPI Filter programming.

The three sections that follow will introduce you to the seven types of ISAPI Filters you can create. You'll get a chance to see how each ISAPI Filter type works and what you can use it for in a production environment. All of these examples are purposely kept simple so that you can see the underlying theory put into practice, rather than get mired in the code usually required to create a fully functional ISAPI Filter.

Understanding ISAPI Filters

We covered the basic theory behind ISAPI in Chapter 8. In this section we'll look at some ISAPI Filter specifics you need to know in order to create ISAPI Filters of your own. The following sections will contain information about the requirements for selecting filter options and the seven kinds of ISAPI Filters you can create using Visual C++. For the most part, this information is required reading before you can create even a simple ISAPI Filter because it helps you understand how the ISAPI Extension Wizard works.

Choosing Filter Options

We saw the first page of the ISAPI Extension Wizard in Chapter 8. In essence, this page allows you to define what kind of ISAPI application you want to create, name the classes that will go into that application, and provide a description for the end result. Page two of the ISAPI Extension Wizard, shown in Figure 9-1, is ISAPI Filter specific. It's very important as far as filter designers are concerned because it contains the options you'll need to set the filter's monitoring options.

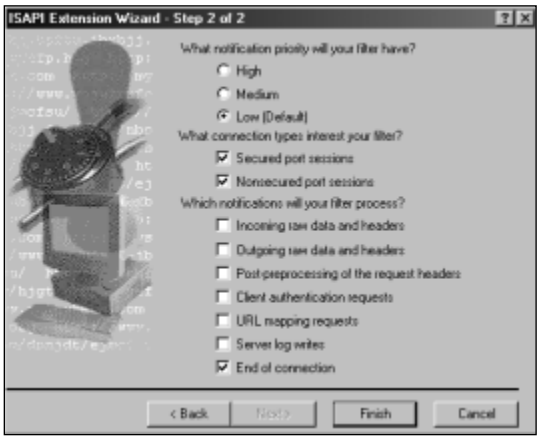

Figure 9-1: Page two of the ISAPI Extension Wizard is ISAPI Filter specific.

Figure 9-1 shows three sections of options. The first section determines the filter's priority. *Low*, the default, works just fine in most cases. You'll want to use this level for any kind of filter that performs a general background task. A security filter may require that you set the filter's priority to *medium*. After all, you don't want the filter to react after the event — a reaction during the event would be much better. Finally, the *high* priority setting should be reserved for emergency-level filters. For example, you might want to send a message out to everyone that the server is going down due to a power failure.

The second section of the ISAPI Extension Wizard - Step 2 of 2 dialog contains two entries. Check the first entry if you want to monitor events when a user has a secure connection. For example, if you plan to provide extra services to a registered user that an anonymous user doesn't get, you may want to check this option. The second option, Non-secured Port Sessions, allows you to monitor events when a user doesn't have a secure connection to the server. That's the way most users will access your site if you have a general Web site on the Internet.

The third section contains a list of notifications (events) that your filter will track. Every time the specified event happens, your filter will get called. However, two

things will affect when it gets called. If you set the priority of your filter to low, then any high- or medium-priority-level filters will get to react to the event first. In addition, your filter has to be set to monitor the event with the user's current security level. In other words, your filter won't get called at all if you set it to monitor non-secured activity and the user is in secured mode.

Choosing between the ISAPI Filter Types

As previously mentioned, you can handle seven phases of Web-server processing with an ISAPI Filter. Each of these seven types corresponds to a specific server event. In other words, an ISAPI Filter can be viewed as a special kind of event handler for Web servers. Table 9-1 shows the seven message types that you can choose by default.

Table 9-1
ISAPI Filter Types by Message

Message Type	*Function*	*Description*
Incoming raw data and headers	OnReadRawData	This filter message type allows you the greatest access to direct client input. You get a chance to see the client input before the server processes it. Of course, this means you'll have to dig through a lot of extraneous information before you find what you want, but you can also be sure that the server hasn't removed anything. This message filter uses the PHTTP_FILTER_RAW_DATA data structure to pass data to your application.
Outgoing raw data and headers	OnSendRawData	This filter message type allows you to see the final output of the server before the client sees it. You can modify the data as needed after the server has done everything normally done to the data stream. Again, you'll go through a lot of extraneous information using this type of filter, but then you can also be sure that the client will see exactly what you want it to see. This message filter uses the PHTTP_FILTER_RAW_DATA data structure to pass data to your application.

Message Type	*Function*	*Description*
Post-preprocessing of the request headers	OnPreprocHeaders	Use this filter message type if you want the server to perform some preprocessing of the incoming data before your filter sees it. In most cases, you'll get to see everything of value that the server will see, without dealing with a lot of low-level information first. The advantage to this particular message filter type is that you don't have to perform a lot of data massaging to get the desired effect. Obviously, you'll have to be certain that the server won't change the data in some way before you see it. This message filter uses the PHTTP_FILTER_PREPROC_HEADERS data structure to pass data to your application.
Client authentication requests	OnAuthentication	Your client will get called as part of the user authentication process if you hook this message. Doing so allows you to create customized authentication routines. You can still call on the standard processing method to complete the task once you complete any custom processing. This message filter uses the PHTTP_FILTER_AUTHENT data structure to pass data to your application.
URL mapping requests	OnUrlMap	Hiding data locations on your server is one reason to hook this message. For example, you may not want a user to know about a virtual directory located on your server, so you redirect the message stream there after the server passes it to you. This process is called mapping a logical address to a physical location on the server. This message filter uses the PHTTP_FILTER_URL_MAP data structure to pass data to your application.

Continued

Table 9-1 *(continued)*

Message Type	*Function*	*Description*
Server log writes	OnLog	There are any number of reasons to write information to the logs on your server. You may want to provide enhanced message information, hide entries for secret locations on your Web server, or just monitor the logs for specific kinds of entries and raise a flag for the administrator or Webmaster. This message filter uses the PHTTP_FILTER_LOG data structure to pass data to your application.
End of connection	OnEndOfNetSession	This filter message type allows you to detect the time when the client disconnects from the server. In essence, it's too late to really do anything for the client except send a goodbye message. However, you can use this hook to perform postprocessing of a session. For example, a user may have uploaded data that needs to be verified and placed in a database. Looking for the end of the connection would be a good time to perform this kind processing. That way your database would always contain up-to-date information. There aren't any data structures associated with this message filter.

Like the ISAPI Extension CHttpServerContext class context pointer, ISAPI Filters get passed a CHttpFilterContext class variable that's a pointer to information about the current client request. You can perform many of the same types of processing using this context pointer as you can perform using the ISAPI Extension equivalent, including using the GetServerVariable( ) method. This pointer also allows you to write data to the client and perform server support functions.

Most ISAPI Filter functions have two arguments, the previously mentioned CHttpFilterContext pointer and access to an event specific data structure as well. This data structure contains information that is specific to the current event when referenced to the current client. In other words, you gain access to the information that the Web server is using to process the client request. Table 9-2 shows the ISAPI Filter function, associated data structure, and a description of that structure.

Table 9-2
ISAPI Filter Data Structures

Function	*Data Structure*	*Description*
OnAuthentication	PHTTP_FILTER_AUTHENT	This data structure contains the user name and password, along with the size of each argument.
OnEndOfNetSession	N/A	The only event that doesn't provide a second argument containing event-specific data.
OnLog	PHTTP_FILTER_LOG	This data structure contains eight of the values you'll normally find in a Web-server log file: client user name, client host name, server name, the current operation taking place (normally GET), the target of the operation, any parameters required to execute the operation, the HTTP status, and the Win32 status. Most log files contain additional information that you gain access to through the context pointer.
OnPreprocHeaders	PHTTP_FILTER_PREPROC_HEADERS	What you'll find in this data structure depends on the operation taking place. There are actually three different contents depending on whether the server is getting, setting, or adding a header to the client response. In all three cases, the structure deals with a filter context, the name of the header in question, and its value.

Continued

Table 9-2 *(continued)*

Function	*Data Structure*	*Description*
OnReadRawData	PHTTP_FILTER_RAW_DATA	This data structure gets used for both the OnReadRawData( ) and OnSendRawData( ) functions. It contains the entire client request or resulting Web page from the server. In other words, this structure represents the extreme limits of input to and output from the Web server. The data structure contains a data buffer, a variable that tells how much data the buffer actually contains, and a variable that tells the total size of the buffer.
OnSendRawData	PHTTP_FILTER_RAW_DATA	See OnReadRawData.
OnUrlMap	PHTTP_FILTER_URL_MAP	This data structure contains a pointer to the original URL requested by the client, a pointer to the physical location of the URL on the Web server's hard drive, and the size of the physical location buffer.

As you can see, each type of ISAPI Filter has a very specific purpose and provides the data required to perform a specific task. Choosing the right ISAPI Filter type is extremely important. Not only does your choice affect the filter's placement within the scheme of data processing, but it also affects the kind of data that the ISAPI Filter will be able to access in an attempt to perform its main function — filtering the data flow between the client and server.

Deciding between ISAPI Filters and ISAPI Extensions

You don't have to spend a lot of time deciding whether to create an ISAPI Filter or an ISAPI Extension (or even both). The differences between the two types of ISAPI

applications are pretty easy to figure out. Once you do so, choosing the one you need becomes fairly simple.

An ISAPI Filter will always react to events on the server itself. A user attempting to log onto your Web site will generate an event that an ISAPI Filter can monitor. You'll normally use it to modify the data flow between the client and server. For example, if your application saw an SF_NOTIFY_AUTHENTICATION event take place, it could display a dialog box for the user to enter a name and password. If these inputs were correct, the user would gain access to the Web site.

ISAPI Extensions are used in the same situations as their CGI and ASP counterparts. For example, if you wanted to create an order entry system, you'd likely use an ISAPI Extension. The ISAPI Extension would receive data from a form that the user filled out, process the information, add it to the database, and finally send some type of receipt back to the user. This is the same process that you would follow when using a script; the only difference is that now you're using a DLL instead. Unlike an ISAPI Filter, an ISAPI Extension doesn't monitor events on the server — it acts in every way like an application would.

Sometimes the choice of an ISAPI Filter or an ISAPI Extension doesn't really matter very much. For example, consider a simple Web counter. Every time someone accesses your Web site, you update the counter to show which number visitor they are. You could call an ISAPI Extension from the Web page to update this counter if so desired, or have a filter monitor the SF_NOTIFY_LOG event on the server and send the information automatically. The choice is a matter of personal taste. However, you'd have to question the rationality of using ISAPI to perform such a simple task. In other words, one of the clues that ISAPI may not be the right choice for a given situation is an inability on your part to be able to choose between an ISAPI Filter and an ISAPI Extension to perform the task.

You'll run into situations in which you need to use both an ISAPI Extension and an ISAPI Filter. For example, you might want to display one Web page for an authorized Netscape user, another for an unauthorized Netscape user, and two others for authorized and unauthorized Internet Explorer users. This situation requires some scripting within the HTML form to detect the browser type, a little bit of work on the part of a filter to detect whether the user is authorized or not, and a little work on the part of an extension to generate the proper page. Using this combination of client scripting and ISAPI Extensions on your server represents one way to fully extend your Web site and make it convenient for everyone to use. Obviously, there are ways to perform this same task using a script — the usage clue here is based on the complexity of the comparison that you want to perform.

In short, ISAPI is a great technology to use when speed is a factor or you have a complex task to perform. The task needs to be clearly defined and you need to decide whether a filter or an extension (or both) is most appropriate for the task at hand. However, it's always important to keep the complexity of creating an ISAPI application in mind. Use scripts when scripting is the best solution for the problem — ISAPI is a heavy-duty tool intended for heavy-duty work.

Writing a Simple ISAPI Filter

It's time to begin creating our first ISAPI Filter. The following sections will take you through the steps required to build a very simple ISAPI Filter. Keep in mind that the goal in this section is to learn the essentials of working with an ISAPI Filter, you probably wouldn't create a filter this simple for a real-world environment.

There are five main steps for creating an ISAPI Filter:

1. Determine what kind of ISAPI Filter you want to create. It's important to decide in advance which server events you want to monitor and how you plan to react to them.
2. Create the ISAPI Filter shell.
3. Write the main ISAPI Filter code.
4. Decide which kind of return value to provide to the server (you may need to provide different values depending on the results of any operations performed within the ISAPI Filter).
5. Perform any required repetitive actions. Some ISAPI Filters loop. The one ISAPI Filter type that always loops is the OnAuthentication( ) handler. We'll discuss the three-loop sequence that you'll always follow when working with this filter type later in the chapter.

One of the easiest ISAPI Filter types to create is the OnLog( ) handler. This particular server event handler places very few restrictions on the programmer. For the most part, you'll find that the OnLog( ) handler is the safest type of ISAPI Filter to experiment with as you learn the basics of working with an ISAPI Filter. With this in mind, our first example will deal with the OnLog( ) handler.

CD-ROM Location: Chapt09\ChngLog
Program Name: ChngLog.DLL
Source Code Modules in Text: ChngLog.CPP

In this example, we'll look for certain log-entry requests to appear based on the Web page that the client is requesting. When the user requests the target log page, the ISAPI Filter will make a change in the server log. You could use this capability to create enhanced or augmented log entries using standard log formats. For example, you might want to include more than just the date and time of the access. The description might include more than just the requested URL.

There's an alternative use for this particular example; you could use it for secure page hiding. The main server log might contain all user requests for public pages, while a secondary log would contain the secure page requests. The main log would only contain a marker indicating that the page request was secure (prompting the administrator to look in the secondary log for an entry).

Obviously, this whole idea is very simplified, but it always pays to look at even the smallest ways of improving security. Remembering that hackers often use log files they find on a server to dig deeper into a company's resources is justification enough, but you'll find other reasons as well for keeping some log entries classified. In some cases, you may even want to completely eliminate certain log entries simply because you don't want to monitor them. Suffice it to say that you'll want to exercise care when using this kind of filter, but it's a good place to start.

Creating the ISAPI Filter Shell

The process for starting the ISAPI Filter is just about the same as the one for creating an ISAPI Extension. The only real difference is the second page of the ISAPI Extension Wizard, which appears in Figure 9-1. The ISAPI Extension Wizard - Page 2 of 2 dialog allows you to define what type of ISAPI Filter you'll create, the priority that it gets executed at, and the types of Web pages that it affects (secure versus non-secure). The following steps will help you create a program shell.

Try This

1. Use the File | New... command to display the New dialog. Select the ISAPI Extension Wizard. (Even though we're creating a filter, you'll still need to use the ISAPI Extension Wizard to do it.)
2. Type the name of the ISAPI Filter you want to create. I used ChngLog for the example ISAPI Filter, but you can use any name you want.
3. Click OK. You'll see the ISAPI Extension Wizard - Step 1 of 2 dialog shown in Figure 9-2. This is where you'll select the various characteristics for your ISAPI Filter. Notice that there are only three main areas to the dialog. You can choose to create an ISAPI Filter by checking the first checkbox and an ISAPI Extension by checking the second checkbox. The third area defines how you'll link MFC to your application.

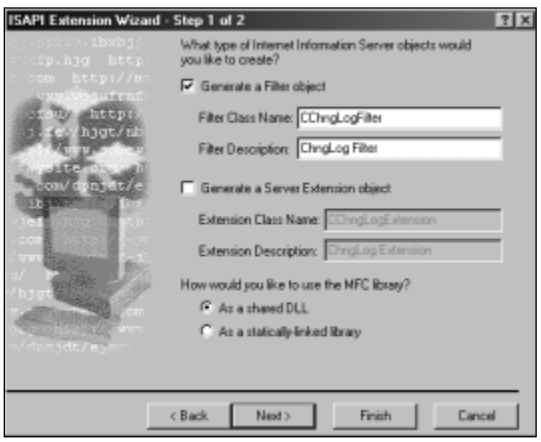
ISAPI Extension Wizard - Step 1 of 2
What type of Internet Information Server objects would you like to create?
Generate a Filter object
Filter Class Name: CChngLogFilter
Filter Description: ChngLog Filter
Generate a Server Extension object
Extension Class Name: CChngLogExtension
Extension Description: ChngLog Extension
How would you like to use the MFC library?
As a shared DLL
As a statically-linked library
< Back
Next >
Finish
Cancel

Figure 9-2: The ISAPI Extension Wizard - Step 1 of 2 dialog allows you to enter a description for your ISAPI Filter.

4. Check the Generate a Filter object option, and uncheck the Generate a Server Extension object option as shown in Figure 9-2. (Notice that adding the filter also adds another step to the process.) We also need to enhance the description. Right now it doesn't say much. You'll want to provide a short, concise statement of what your ISAPI does. The description appears as a string that you can use within the DLL as needed. This description won't show up in the Properties dialog when someone opens it for your ISA, so you'll want to add some additional text to the version information for your DLL as well.
5. Type, **Classify some log entries for security reasons.** in the Filter Description field.
6. Click Next. You'll see the ISAPI Extension Wizard - Step 2 of 2 dialog shown in Figure 9-1. This is the page you'll use to select the events and type of monitoring your filter will provide. You have three areas to consider. See the "Choosing Filter Options" section to get more details about these options.
7. Check the Server Log Writes checkbox, and uncheck the End of Connection checkbox. We want to activate our filter when the user requests specific kinds of access to the server since we're creating a simple filter to keep some log entries classified.
8. Click Finish. You'll see a New Project Information dialog like the one shown in Figure 9-3. Make sure you double-check all of the settings for your filter because the New Project Information page provides a detailed breakdown of the events that the filter will get to see.
9. Click OK. The ISAPI Extension Wizard will create the required program shell for you.

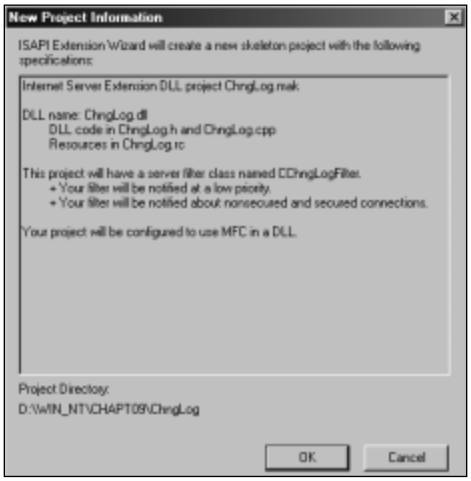
New Project Information
ISAPI Extension Wizard will create a new skeleton project with the following specifications:
Internet Server Extension DLL project ChngLog.mak:
DLL name: ChngLog.dll
DLL code in ChngLog.h and ChngLog.cpp
Resources in ChngLog.rc
This project will have a server filter class named CChngLogFilter.
+ Your filter will be notified at a low priority.
+ Your filter will be notified about nonsecured and secured connections.
Your project will be configured to use MFC in a DLL.
Project Directory:
D:\WIN_NT\CHAPT09\ChngLog
OK
Cancel

Figure 9-3: The New Project Information dialog shows the type of ISAPI Filter you'll create.

Writing the ISAPI Filter-specific Code

The coding portion of this example is short and to the point. Filters can quickly get out of hand in the complexity department, and debugging them is fairly difficult. In most cases, you'll want to make your filter programs as short as possible. Modularization is a big help as well when it comes time to troubleshoot a faulty filter. Listing 9-1 shows the code you'll need to add to the OnLog( ) method to make it work.

Listing 9-1: **OnLog( ) Method Source Code**

```
DWORD CChngLogFilter::OnLog(CHttpFilterContext *pCtxt, PHTTP_FILTER_LOG pLog)
{
// Check for a specific notification.
    if (strstr(pLog->pszTarget, "NoLog.htm") != NULL)
    {
        // Change the user entries.
        pLog->pszClientHostName = "Classified Host";
        pLog->pszClientUserName = "Classified User";

        // Change the server entry.
        pLog->pszServerName = "Classified Server";

        // Change the operation.
        pLog->pszOperation = "Classified Operation";

        // Change the target information to Classified.
        pLog->pszTarget = "Classified Target";
        pLog->pszParameters = "Classified Parms";

        // Change the status words--could be a log file number.
        pLog->dwHttpStatus = 999;
        pLog->dwWin32Status = 999;
    }

    // Pass control to the next filter.
    return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

As you can see, we'll simply monitor the log entries for a particular target, which could be any number of things but is normally another Web page. Once we find the log entry we're looking for, we'll change two members, pszTarget and pszParameters, to their classified setting.

Loading the Filter

At this point, you can compile the new ISAPI Filter. However, even after you move it to the Controls or Scripts directory on your server, you'll still have to perform one other task. Unlike an extension, an ISAPI Filter gets loaded when you start or update the service.

For the examples in this chapter, you should be using Windows NT Server Version 4.0 or above, with a minimum of Service Pack 3 and IIS Version 4 installed. Older versions of these products will require you to perform some steps manually. In addition, you'll see differences in the screen shots in this chapter and the ones you'll see on the target server. To install an ISAPI Filter on an older version of Windows NT Server, you'll need to stop the target service, make a registry entry, and then start the service up again. The filter will get loaded as part of the starting process. The WWW service stores its filter entries in the following registry value: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters \Filter DLLs. You'll probably find one or more values in this location already. All you need to do is add a comma and then type the location of your new ISAPI Filter. Make absolutely certain you perform this step before going on to the next section or the filter won't load. In fact, this is the setting you'll want to check first if you have trouble getting the filter to work.

Users of IIS version 4.0 and above will really appreciate the fact that they no longer have to edit the registry manually to load an ISAPI Filter. The following procedure shows you how to add an ISAPI Filter to the Default Web Site using Microsoft Management Console.

Try This

1. Open the Microsoft Management Console. You'll see a window like the one shown in Figure 9-4. (This is the window you'll see if you choose the Internet Service Manager option on the Start menu.)
2. Right-click the Web site that you want to add the filter to, and choose Properties from the context menu. You'll see a Web Site Properties dialog. The example uses the Default Web Site and the screen shots will reflect this fact.
3. Select the ISAPI Filters tab and you'll see a Web Site Properties dialog like the one shown in Figure 9-5.

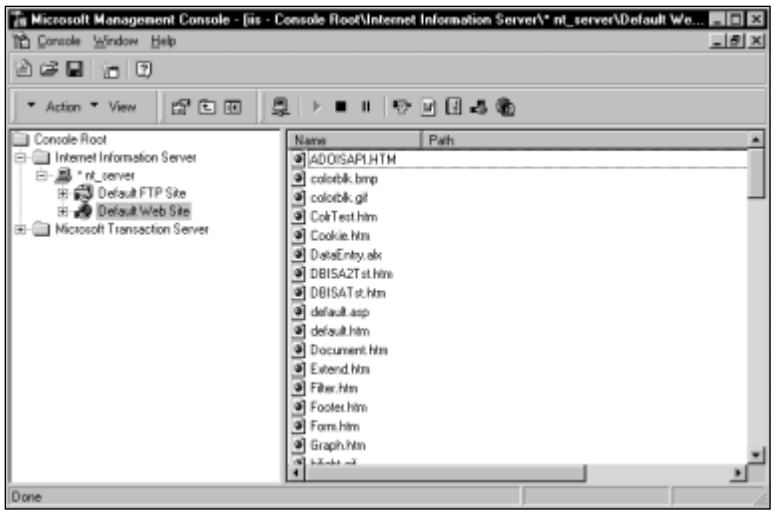

Figure 9-4: The Microsoft Management Console allows you to manage the entire server, including Internet services.

Figure 9-5: The ISAPI Filters tab of the Web Site Properties dialog allows you to add new filters to IIS.

4. Click the Add button and you'll see a Filter Properties dialog like the one shown in Figure 9-6.

Figure 9-6: The Filter Properties dialog allows you to define the name and executable filename of a new ISAPI Filter.

5. Type the human readable name in the Filter Name field. In this case, I used a name of Classified Log Entries.
6. Use the Browse button to find the ChngLog.DLL file (or whatever ISAPI Filter you want to add to the Web site). Once you select the file, its name will appear in the Executable field.
7. Click OK once to close the Filter Properties dialog.
8. Click OK to close the Web Site Properties dialog. IIS will automatically load the filter for you. At this point, we need to check whether the ISAPI Filter got loaded successfully.
9. Reopen the ISAPI Filters tab of the Web Site Properties dialog. Figure 9-7 shows what a successful load of an ISAPI Filter looks like. Notice that the Status column contains an up arrow and that the Priority column contains the priority of the ISAPI Filter.

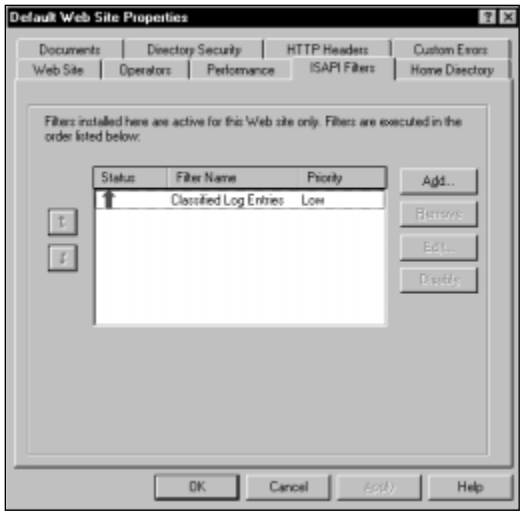

Figure 9-7: Once you load the ISAPI Filter, you need to check to verify that it's operational by looking at the Status and Priority columns.

Testing the ISAPI Filter

By now you should have your ISAPI Filter completed and compiled. You should have placed it in a location where you normally keep filters — which could be a Controls, Scripts, or even the System32 directory. Remember to load the ISAPI Filter, then verify that it's actually operational. (When using an older version of IIS, you'll need to fiddle with the registry as noted in the "Loading the Filter" section of this chapter.) Now we're ready to test the filter out.

The first thing we'll need are three Web pages. The first is a central Web page we'll use to switch between the two others — one logged normally and the other logged as classified. Call the first Web page Filter.HTM. Listing 9-2 shows the source code for this part of the test setup.

Listing 9-2: **Main ISAPI Filter Web Test Page Source Code**

```
<HTML>
<HEAD>
<TITLE>ISAPI Filter Test</TITLE>
</HEAD>
<BODY>

<!-Display a heading.->
<H3><CENTER>ISAPI (NoLog) Filter Tester</CENTER></H3>

<!-Display the links and some explanatory text.->
Choosing the first link will create a classified log entry,
while the second one creates a standard log entry.<P>

Log this Link as <A HREF="NoLog.htm">Classified</A><P>
Log this Link <A HREF="LogIt.htm">Normally</A>

</BODY>
</HTML>
```

There shouldn't be anything too mysterious about this code. We've used it all before. All this page contains is a heading, some explanatory text, and a couple of links. We'll use it to switch between the two test pages without going through a lot of extra work.

The names for the two test pages are important; you must name the first test page NoLog.HTM or the filter won't work. (Because it's looking for a target by this name, you have to provide one for testing purposes.) Listing 9-3 contains the source code

for this page. The second text page is named LogIt.HTM. Listing 9-4 contains the source code you'll need to create it. Both test pages contain a header and some text, but that's about it.

Listing 9-3: **Not-Logged Web Test Page Source Code**

```
<HTML>
<HEAD>
<TITLE>ISAPI Filter Test</TITLE>
</HEAD>
<BODY>
<H3><CENTER>This Page is <EM>Not</EM> Logged</CENTER></H3>
Which means that you won't see any of the normal entries in the log.<P>
What you will see are a bunch of classified entries instead.
</BODY>
</HTML>
```

Listing 9-4: **Logged Web Test Page Source Code**

```
<HTML>
<HEAD>
<TITLE>ISAPI Filter Test</TITLE>
</HEAD>
<BODY>
<H3><CENTER>This Page is Logged</CENTER></H3>
Which means that you'll see all the normal entries in the log.
</BODY>
</HTML>
```

We're all ready to check out the ISAPI Filter. All you need to do is open the Filter.HTM Web page using your Web browser. Make absolutely certain you access the Web server through an intranet connection. Double-clicking on the Filter.HTM file in Explorer and opening it as a file won't force the Web server to make the appropriate log entries. Once you've gone between the two test pages several times, open the log file. If you have your Web server set up like I do, there's one log file for each day in the System32\LogFiles directory on your Web server. Here's what the log entries looked like when I tried the ChngLog ISAPI Filter out using the Microsoft IIS log file format (IIS provides several log file formats — each of which differs slightly in appearance).

You'll have to stop the Web server before you attempt to look at the log file. Otherwise, Windows NT will report that the log file is locked by another service. Make sure you restart the service before you proceed to other examples in the book.

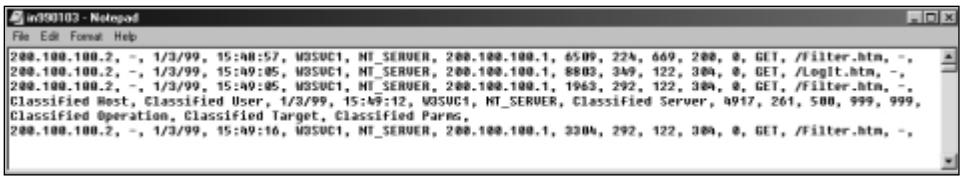
in390103 - Notepad
File Edit Format Help
200.100.100.2, -, 1/3/99, 15:48:57, W3SVC1, NT_SERVER, 200.100.100.1, 6509, 224, 669, 200, 0, GET, /Filter.htm, -,
200.100.100.2, -, 1/3/99, 15:49:05, W3SVC1, NT_SERVER, 200.100.100.1, 6803, 349, 122, 304, 0, GET, /LogIt.htm, -,
200.100.100.2, -, 1/3/99, 15:49:05, W3SVC1, NT_SERVER, 200.100.100.1, 1963, 292, 122, 304, 0, GET, /Filter.htm, -,
Classified Host, Classified User, 1/3/99, 15:49:12, W3SVC1, NT_SERVER, Classified Server, 4917, 261, 500, 999, 999,
Classified Operation, Classified Target, Classified Parms,
200.100.100.2, -, 1/3/99, 15:49:16, W3SVC1, NT_SERVER, 200.100.100.1, 3304, 292, 122, 304, 0, GET, /Filter.htm, -,

Figure 9-8: An example log file after the test ISAPI Filter has modified the entries.

The fourth line in Figure 9-8 actually word wraps to the fifth line so that you can see the entire entry. This entry would take just one line in the actual log file.

Notice that all of the log entries show the appropriate Web page except the ones for NoLog.HTM. The only NoLog.HTM entry in the illustration is on the fourth line down from the top. These entries are marked as classified because the ChngLog filter changed them. On the other hand, the LogIt and Filter entries appear just as they should. Notice that the other entries provide enough information for a hacker to break into the system, while the classified entries provide very little usable information. Obviously, all this was done automatically — the filter responded to the log event in the background.

Working with Post-preprocessing Filters

The very first stage of processing that the server performs is to separate the raw input from the client into several areas. One of the most important data areas is the request header because it defines how processing will proceed. That's the area of the client request that a post-preprocessing ISAPI Filter works with — the request header. The following sections will help you understand what the request header contains and how it can help you better comply with user requests.

Once you have a better understanding of the request header, we'll look at an example that shows the request header in action. You can use this example as a diagnostic aid when working on your own ISAPI Filters. Often an ISAPI Filter is doing exactly what it's supposed to do, but doesn't work as anticipated because the client request is wrong. This diagnostic aid can help you detect these types of client request errors (especially in scripts) and fix them.

Understanding Request Headers

The various standards committees on the Internet have provided a wealth of request types that a client can make. You'll also find that various browsers have special requests that they support. It doesn't take too long to figure out that you could literally spend all your time monitoring requests from various users to find just one piece of information that you actually need. Fortunately, there's a way to make things a bit easier if all you plan to do is to maintain statistics or provide

some type of header enhancement. The following list describes the three standard pieces of information that you'll always get from a client:

- **Method:** A method usually defines what the client wants to do. Get and post are the most common methods you'll see for a client accessing a Web page. You'll also find that this standard part of the request header is the most flexible from the client's perspective, since servers need to respond to more than a few standard methods.
- **URL:** Every request has to provide a URL of some kind; otherwise, the server can't provide any information in return.
- **Version:** Several versions of HTTP are out now, so it's important to know what version the browser is using. The server normally needs to avoid using special responses with browsers that support older versions of HTTP.

A request header can contain a lot of other information; it just depends on what kind of service the client is requesting. The least complex request headers are for Web services. We'll examine these types of request headers in this section of the chapter. News server and mail server requests become quite a bit more complex because the request header needs to include things like from and date request headers. Here's a list of the standard request headers (although many more exist and you can use a lot of different options to modify the results produced with the ones in this list):

- **From:** The user's name (normally used with mail requests or for logging purposes). There's no guarantee that the name is actually accurate, so you can't use it for security or authentication purposes.
- **Accept**: Provides a list of content types that the client can accept from the server. Each content type is separated by a comma. Parameters for a given content type are separated by a semicolon. Typical content types include audio/mpeg for MPEG audio files, and image/jpeg for JPEG images. You may also see wildcard combinations like *.* for all files or audio/* for all audio file formats. Every browser must accept the text/plain and text/html content types at a minimum.
- **Accept-Encoding:** This header is similar to the Accept request header, but it defines content encoding types. A typical encoding type would be x-zip for files using the ZIP compression method.
- **Accept-Language:** This header is similar to the Accept request header, but it defines the preferred language. Language examples include en for generic English, de for generic German, and en_UK for British English.
- **User-Agent:** The browser will identify itself in this request header. You'll normally see the product name, followed by a slash and the version number.

- **Referer:** In some cases, you'll see this request header that identifies the Web site that referred the user to your site. It comes in handy for determining whether an advertising banner or link on another site is having much of an effect on your site. This request header can also be used to help find bad links or to perform back-tracing to see how a user searches for your site.
- **Authorization:** You'll find security information in this request header. There isn't a specific format because every security scheme uses a different method of presenting itself. Some authentication methods are as simple as a user name and password passed in clear text. Check the documentation for your particular authentication agent for details on what you'll find in this particular request header.

Learn about basic authentication at: `http://www.w3.org/History/1993/WWW/AccessAuthorization/`. This site provides a complete overview along with links to other pertinent information about basic authentication. Pretty Good Privacy (PGP) maintains a site at: `http://pilot.msu.edu/user/heinric6/pgp.htm`. Not only will you find out about this authentication method, but you can also get details on two books that describe PGP. Kerberos, a newer authentication method, is explained at: `http://www.cisco.com/warp/public/106/`. Another good place to look for general Kerberos information is: `http://www.con.wesleyan.edu/~triemer/network/kerberos/kerberos_tcp.html`. This site includes both white paper and newsgroup links. Obviously, the Microsoft documentation for Windows 2000, any associated Web sites, and Windows 2000-specific newsgroups will also contain information about the Microsoft version of Kerberos.

- **Charge-To:** (or ChargeTo:) Some sites charge for the information they provide. This request header tells whom to charge for services. The first word of this request header always contains the specification of the namespace providing the charging authority. The exact format of the rest of the header is determined by the charging authority.
- **If-Modified-Since:** A browser or proxy server will use this request header if it simply wants to see if the data on a site has changed since a given date. If the data hasn't changed, then the server sends a 304 (not modified) reply in place of the more familiar 200 (OK) reply. The server won't download any data to the browser if the data hasn't changed since the specified date.
- **Pragma:** This request field is similar to a macro. It simply tells the server to perform a specific task. The only Pragma: entry that's defined right now is no-cache. It tells a proxy server to download a new copy of a page from the Web server even though the local content hasn't expired yet. Other Pragma: values are possible, but both server and client would need to be designed to understand them.

Every request header consists of one word (multiple words can be linked into a single word using hyphens or underscores) followed by a colon. For example, the from request header that I mentioned earlier looks like this: From: <User Name>. The single word followed by a colon is a standard method of identifying a new request header, which means that you'll want to avoid using this syntax for standard text in any response headers that you create, or request headers that you modify. You'll use this syntax when asking to look at a specific request header (the three standard headers that I mentioned earlier have special key words).

Taking a Look at the Client's Input

Figuring out what kind of information the client is requesting is the number one priority of any Web server. After all, you can't answer the question if you don't understand it. Essentially, the client and the server are holding a conversation. Good communication is essential if you want to provide what the user wants in a timely manner.

In the following sections, we look at how you can view the request header information that the client sends to the server. I'm using the most common request headers because it allows you to see what the client is sending no matter what the request is. These three request headers even have special Visual C++ names: method, URL, and version. Unlike most header requests, you won't need to add the colon for these three request headers.

CD-ROM Location: Chapt09\Req1
Program Name: Req1.DLL
Source Code Modules in Text: Req1.CPP

Creating the Req1 ISAPI Filter Shell

Let's create the filter. We'll use the same basic technique that I've outlined earlier in this chapter. The following steps will explain some differences from that procedure that we'll need for this particular example.

Try This

1. Create a basic ISAPI Filter using the ISAPI Extension Wizard. I've given mine the name of Req1, but you can use any name you like.
2. Check the ISAPI Filter option and uncheck the ISAPI Extension option on the first page of the MFC Extension Wizard.
3. Choose to monitor both secured and nonsecured port sessions.
4. Check the Post-preprocessing of the request headers option and uncheck the End of connection option. Figure 9-9 shows what the New Project Information dialog should look like for this example.

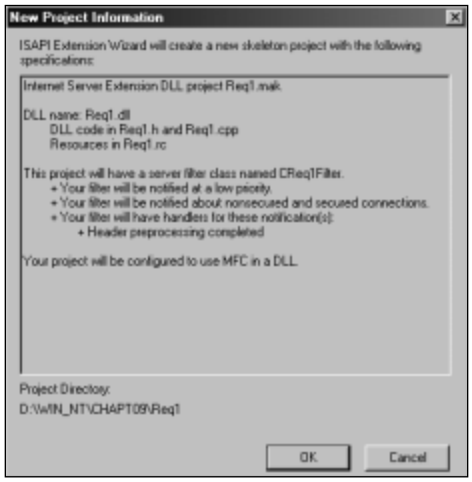
New Project Information
ISAPI Extension Wizard will create a new skeleton project with the following specifications:
Internet Server Extension DLL project Req1.mak
DLL name: Req1.dll
DLL code in Req1.h and Req1.cpp
Resources in Req1.rc
This project will have a server filter class named CReq1Filter.
+ Your filter will be notified at a low priority.
+ Your filter will be notified about nonsecured and secured connections.
+ Your filter will have handlers for these notification(s):
+ Header preprocessing completed
Your project will be configured to use MFC in a DLL.
Project Directory:
D:\WIN_NT\CHAPT09\Req1
OK
Cancel

Figure 9-9: The New Project Information dialog for the Req1 example.

Adding Some Code to Req1

Now that you have a basic shell put together, let's create some code for it. Listing 9-5 shows the code you'll add to the OnPreprocHeaders( ) function. It also shows the results of the various settings we made. I'll explain how the code works after the listing.

Listing 9-5: **OnPreprocHeaders( ) Method Source Code**

```
DWORD CReq1Filter::OnPreprocHeaders(CHttpFilterContext* pCtxt,
   PHTTP_FILTER_PREPROC_HEADERS pHeaderInfo)
{
   CString   oBuffer;   // Buffer for client output.
   DWORD   dwSize;      // Size of the buffer.
   LPVOID   pvOutput;   // Client output.

   // Send a response header so that the browser will display our data.
   pCtxt->ServerSupportFunction(SF_REQ_SEND_RESPONSE_HEADER, NULL, NULL, NULL);

   // Create the usual header.
   oBuffer = "\r\n<HTML>\r\n<HEAD>\r\n<TITLE>Request Headers Test
Page</TITLE>\r\n</HEAD>\r\n<BODY>";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);
```

Continued

Listing 9-5: *(continued)*

```
   // Display a page heading.
   oBuffer = "\r\n<CENTER><H2>Post-preprocessing Request Headers
Example</H2></CENTER>";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);

   // Display the requested URL.
   oBuffer = "\r\nRequested URL (No Protocol or Server Name Attached): ";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);
   pHeaderInfo->GetHeader(pCtxt->m_pFC, "url", pvOutput, &dwSize);
   dwSize = strlen(LPCTSTR(pvOutput));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);

   // Display the requested method.
   oBuffer = "<P>\r\nRequested Method: ";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);
   pHeaderInfo->GetHeader(pCtxt->m_pFC, "method", pvOutput, &dwSize);
   dwSize = strlen(LPCTSTR(pvOutput));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);

   // Display the requested version.
   oBuffer = "<P>\r\nRequested Version: ";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);
   pHeaderInfo->GetHeader(pCtxt->m_pFC, "version", pvOutput, &dwSize);
   dwSize = strlen(LPCTSTR(pvOutput));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);

   // Create the usual footer.
   oBuffer = "\r\n</BODY>\r\n</HTML>";
   dwSize = oBuffer.GetLength();
   pvOutput = LPTSTR(LPCTSTR(oBuffer));
   pCtxt->WriteClient(pvOutput, &dwSize, 0);

   // return the appropriate status code
   return SF_STATUS_REQ_FINISHED_KEEP_CONN;
}
```

We've used a couple of new techniques in this example. For one thing, the structure sent to the OnPreprocHeaders( ) function doesn't contain any data. What it does contain is three function calls that you can use to obtain data from the server. This

distinction is very important because it's the first time we've had to call a function in this manner.

This filter is also different from the first one we created in another way; it will actually create a Web page and send it to the client. The process for outputting a Web page from an ISAPI Filter is similar to an ISAPI Extension, but the methodology is completely different.

First, we create a standard response header for the client. We could have added some additional response header information, but it really isn't important to do so for this example. Any client will be able to react to the standard HTML text we'll be sending.

The next thing you'll see is code for sending some standard HTML to the client. All we're doing is creating the header and title area. We also begin the body area so that the client can display the contents of the request header for us.

At this point, we have some headers in place, and we're ready to display the contents of the request header. Take a look at this line of code:

```
pHeaderInfo->GetHeader(pCtxt->m_pFC, "url", pvOutput, &dwSize);
```

pHeaderInfo is the structure passed to our function by the server. There are three function calls within this structure: GetHeader( ), SetHeader( ), and AddHeader( ). We'll use the GetHeader( ) function to retrieve the value associated with a particular request header. It's important to note that you can't retrieve all the request headers at once; you have to get them one at a time. It's also important to understand that besides the three standard request headers, you need to know the exact names of the request headers you want from the client and that the client is under no obligation to supply these request headers.

The GetHeader( ) function needs to know which filter context is making a request. The only way it can identify the filter context is if you pass the request header structure as the first parameter. The m_pFC variable provided with the CHttpFilter class is specially designed for this purpose. The next parameter shows which request header we want; in this case, we want to get the URL request header. pvOutput is the buffer used to store the contents of the request header. dwSize contains the size of the buffer.

We'll call the GetHeader( ) function three times to get all three of the standard request headers and display them on the client using the WriteClient( ) function. The very last thing we have to do is to end the Web page and output it to the client.

Testing the Req1 ISAPI Filter

The first thing you'll need to do to test the Req1 ISAPI Filter is to load it. We looked at that process in the "Loading the Filter"section of this chapter, so I won't explain it again here. Make sure you also unload any other filters you may have loaded previously to keep them from altering the output from the Req1 example ISAPI

Filter. Remember that the filters are placed in a chain and that the output from one filter can affect the input to another. Once you have the filter loaded, you can request a Web page from your Web server. Figure 9-10 shows the results of requesting the Default.HTM page from my test server; your results may vary slightly from the ones shown here.

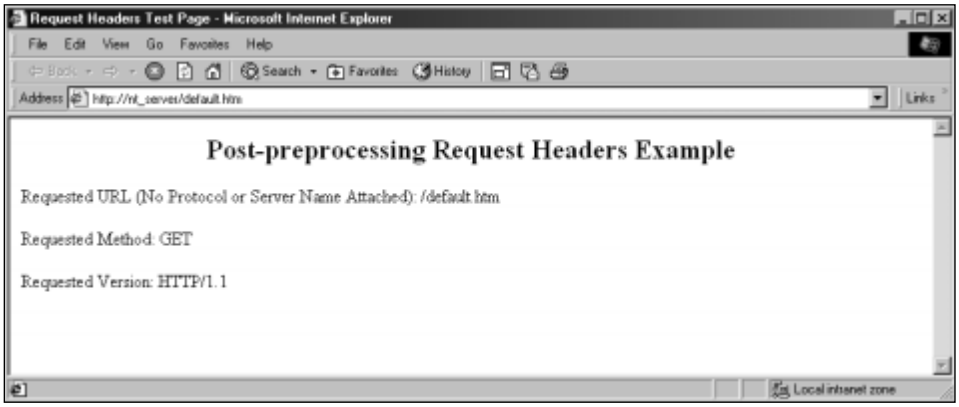
Request Headers Test Page - Microsoft Internet Explorer
File Edit View Go Favorites Help
Search Favorites History
Address http://nt_server/default.htm
Links
Post-preprocessing Request Headers Example
Requested URL (No Protocol or Server Name Attached): /default.htm
Requested Method: GET
Requested Version: HTTP/1.1
Local intranet zone

Figure 9-10: The GetHeader( ) function allows you to see what the client has sent as a request header to the server.

Working with Client Authentication Filters

The affect of security on a Web site can't be overemphasized. Do it right, and your site will likely experience few problems from crackers and others that would try to steal or damage valuable data. Do it wrong, and you'll experience nothing but trouble from various sources. Depending on how bad your security is, anyone from a novice cracker to a disgruntled employee could try to damage information contained on your server.

Of course, good security doesn't just protect you from damaged files. Even more gruesome are the things that various people could do besides damaging your data. Some of the more successful crackers today do nothing to draw attention to themselves, carefully avoiding making themselves known. Their sole purpose in life is to steal your data and sell it to someone else, perhaps a competitor that would just love to know your trade secrets.

Since the media and many other books on the market have probably scared you to death already, let's look at something you can do to protect yourself. Security filtering can provide some additional support for your Web site. Depending on your

programming skills, you can either augment existing technology or even come up with some type of security scheme of your own.

Before you get started working with the example in this section of the chapter, we need to take care of a few housekeeping details. The "Setting Your Security" section below will help you take care of those housekeeping details. Please pay careful attention to everything said in this section.

The example in the following section of the chapter will show you how to create a security diagnostic filter. By looking at the values you receive from the client at the server, this diagnostic will tell you whether you've created the kind of filter you were looking for. It's important to know if your encryption and decryption schemes are working as intended. We'll also look at what you'd need to do to change this example into a simple security filter.

Setting Your Security

IIS comes configured in the way that Microsoft deems best for the average user. If you're using Internet Explorer as a browser, then you probably do have the best possible setup for a production environment. However, the examples in this chapter really aren't designed for a production environment, they're designed to show you how to create filters that will eventually do something in a production environment. With that in mind, we have to make a few changes to the setup that you wouldn't normally make.

Stop right now if you haven't been working on a test machine. The example in this section of the chapter will not only make your Web server nonfunctional for users who want to log on, it will also disable at least some of the security measures you have in place on a production machine. Make absolutely certain that you try this example on a test machine only, not on a production machine.

Now that we have the requisite cautions taken care of, we can take a look at what you need to get started on the example in this chapter. The following procedure will show you how to set the security on your sever so that the example will work. Make sure that you record any settings that you already have in place so that you can restore them later. (Following the procedure in reverse won't net a server with the same security setup that you started with).

1. Open the Microsoft Management Console.
2. Right-click on the Default Web Site entry, and then choose Properties from the context menu.
3. Click the Directory Security tab. Your display should look like the one shown in Figure 9-11.

Figure 9-11: The Directory Security tab allows you to choose the security options for the selected directory (or all directories if you choose the root directory).

4. Click the Edit button in the Anonymous Access and Authentication Control section of the dialog. You'll see an Authentication Methods dialog like the one shown in Figure 9-12.

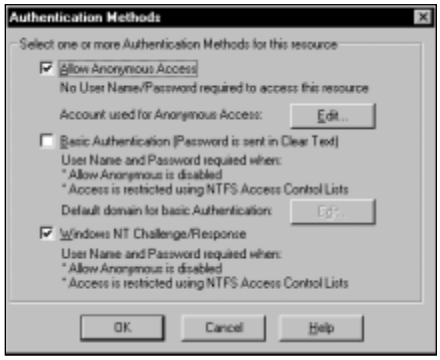
Authentication Methods
Select one or more Authentication Methods for this resource
Allow Anonymous Access
No User Name/Password required to access this resource
Account used for Anonymous Access:
Edit...
Basic Authentication (Password is sent in Clear Text)
User Name and Password required when:
* Allow Anonymous is disabled
* Access is restricted using NTFS Access Control Lists
Default domain for basic Authentication:
Edit...
Windows NT Challenge/Response
User Name and Password required when:
* Allow Anonymous is disabled
* Access is restricted using NTFS Access Control Lists
OK
Cancel
Help

Figure 9-12: The Authentication Methods dialog allows you to choose how clients will get authenticated by IIS.

5. Uncheck both the Allow Anonymous Access and Windows NT Challenge/Response checkboxes. This will prevent Internet Explorer from using the Windows NT Challenge/Response option if you use that browser. Unchecking the Allow Anonymous Access option means that everyone will need to provide a name and password.

6. Check the Basic Authentication checkbox. Both Internet Explorer and Netscape products will send passwords and user names in clear text — which is perfect for test purposes. You'll see the dialog shown in Figure 9-13.

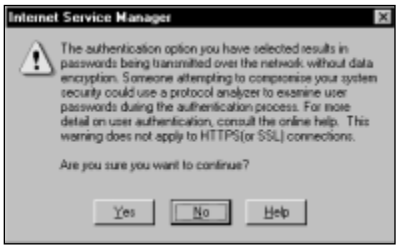
Internet Service Manager
The authentication option you have selected results in passwords being transmitted over the network without data encryption. Someone attempting to compromise your system security could use a protocol analyzer to examine user passwords during the authentication process. For more detail on user authentication, consult the online help. This warning does not apply to HTTPS(or SSL) connections.
Are you sure you want to continue?
Yes
No
Help

Figure 9-13: The Internet Service Manager dialog tells you what security risks are involved in using Basic Authentication.

7. Click Yes when the Internet Service Manager asks whether you're sure that you want to use Basic Authentication. Make sure you read the information on this dialog. Without SSL, anyone can use a product like Sniffer to detect the user names and passwords passed from the client to the server during authentication, and then use that information to hack into your system.
8. Click OK to close the Authentication Methods dialog.
9. Click OK to close the Default Web Site Properties dialog.

At this point, you're ready to work with the examples in the following section of the chapter. To follow this procedure you have to use a test server without any certificates installed. Make sure you disable SSL support if you have a certificate installed, or the examples may not work as expected.

Creating a Security Diagnostic

While this little security diagnostic won't work in every situation, I find that it's very helpful when testing new encryption and decryption code. It will only display the user's name and password in plain text. However, the simplicity of this particular filter is one of its better features. You get a simple indicator of whether your code works or not.

Before we actually create the example, I'd like to cover a few bits of information you need to understand what we're doing. Every ISAPI security filter has to perform the four steps shown below:

1. Ask the user for input by creating an error situation.
2. Determine that the user has provided input.
3. Determine if the user is authorized to use the resource requested.
4. Take some action that either allows the user to use the resource or denies access to it.

Once a filter has determined that the user has provided some input, it can simply pass control to the server for verification. Of course, this isn't always what you want. You may want to perform some additional processing within the filter that the server doesn't normally do. The filter may also have to provide some additional

functionality in the form of decryption or perhaps even specialized error-message handling. The following list gives you some ideas about the additional features that a security filter can provide.

Decryption — You can always choose to encrypt the password and user name at the client end. This means decrypting it at the server end as well. An ISAPI Filter is perfect for this purpose.

Authentication — Windows NT doesn't force you to use the built-in authentication. You can always choose to authenticate the user yourself and give them an appropriate level of security. For that matter, you can perform an authentication, then assign the user to a group and let Windows NT take care of the details. There are a number of different authentication scenarios.

Custom Error Handling — You may not like the way that Windows NT handles errors. An ISAPI Filter can provide at least limited error handling. Of course, the precise level of error handling you provide is determined by the client as well as the server, which means you'll want to have control over which browser the user uses for access.

Specialized Access Handling — There are situations when you may not want everyone to have 24-hour access to your Web site. Neither IIS nor Windows NT currently provides any method for shutting down your Web server to the average user, while keeping it open for everyone else. An ISAPI Filter can provide access based on the current time or any other criteria for that matter. You may even restrict access to certain sites based on server load.

Now that you have a better idea of what this example can do, let's take a look at the example itself. The following section will show you how to create a simple security filter that displays a Web page showing the user's input when asked for a clear text name and password. Remember that the goal here is to look at a simple instance of a diagnostic.

CD-ROM Location: Chapt09\SendErr
Program Name: SendErr.DLL
Source Code Modules in Text: SendErr.CPP

Creating the SendErr ISAPI Filter Shell

Let's begin creating the filter. We'll use the same basic technique that I've shown earlier in the chapter. The following steps will outline some differences from the previous procedure that we'll need for this particular example.

1. Create a basic ISAPI Filter using the ISAPI Extension Wizard. I gave mine the name of SendErr, but you can use any name you like.
2. Check the ISAPI Filter option and uncheck the ISAPI Extension option on the first page of the MFC Extension Wizard.
3. Chose to monitor both secured and nonsecured port sessions.
4. Check the Client authentication requests option and uncheck the End of connection option. Figure 9-14 shows the New Project Information dialog for this project.

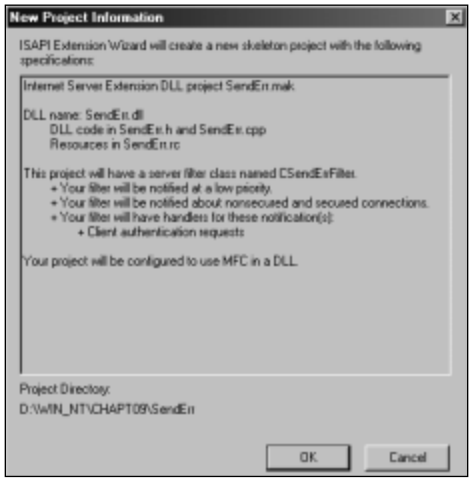
New Project Information
ISAPI Extension Wizard will create a new skeleton project with the following specifications:
Internet Server Extension DLL project SendErr.mak
DLL name: SendErr.dll
DLL code in SendErr.h and SendErr.cpp
Resources in SendErr.rc
This project will have a server filter class named CSendErrFilter.
+ Your filter will be notified at a low priority.
+ Your filter will be notified about nonsecured and secured connections.
+ Your filter will have handlers for these notification(s):
+ Client authentication requests
Your project will be configured to use MFC in a DLL.
Project Directory:
D:\WIN_NT\CHAPT09\SendErr
OK
Cancel

Figure 9-14: The New Project Information dialog for the SendErr ISAPI Filter

Adding Some Code to the SendErr ISAPI Filter

Now that you have a basic shell put together, let's create some code for it. Listing 9-6 shows the code you'll add to the OnAuthentication( ) function. It also shows the results of the various settings we made. I'll explain how the code works after the listing.

Listing 9-6: **OnAuthentication( ) Method Source Code**

```
DWORD CSendErrFilter::OnAuthentication(CHttpFilterContext* pCtxt,
   PHTTP_FILTER_AUTHENT pAuthent)
{
   CString    oBuffer;    // Buffer for client output.
   DWORD   dwSize;        // Size of the buffer.
   LPVOID   pvInOut;      // Client input or output.

   // See if we're getting an anonymous request.
   if (strlen(pAuthent->pszUser) == 0)
   {

      // Get the user's name.
      dwSize = 100;
      pvInOut = " ";
      pCtxt->GetServerVariable("REMOTE_USER", pvInOut, &dwSize);
      dwSize = strlen(LPTSTR(pvInOut));

      if (strncmp(LPTSTR(pvInOut), " ", 1) == 0)
      {

         // Set an error condition.
         pCtxt->ServerSupportFunction(SF_REQ_SEND_RESPONSE_HEADER,
            "401 Access Denied",
            NULL,
            NULL);
      }
      else
      {

         // Indicate the user has supplied a password.
         pCtxt->ServerSupportFunction(SF_REQ_SEND_RESPONSE_HEADER,
            NULL,
            NULL,
            NULL);

         // Store the user's name.
         strncpy(pAuthent->pszUser, LPTSTR(pvInOut), dwSize);

         // Get and store the user's password.
         dwSize = 100;
         pvInOut = " ";
         pCtxt->GetServerVariable("AUTH_PASS", pvInOut, &dwSize);
         dwSize = strlen(LPTSTR(pvInOut));
         strncpy(pAuthent->pszPassword, LPTSTR(pvInOut), dwSize);
      }

      // return the appropriate status code
      return SF_STATUS_REQ_NEXT_NOTIFICATION;
   }
```

```
   // Otherwise, the user must have supplied a name.
   else
   {

      // Send a response header so that the browser will display our data.
      pCtxt->ServerSupportFunction(SF_REQ_SEND_RESPONSE_HEADER, NULL, NULL,
NULL);

      // Create the usual header.
      oBuffer = "\r\n<HTML>\r\n<HEAD>\r\n<TITLE>Password
Check</TITLE>\r\n</HEAD>\r\n<BODY>";
      dwSize = oBuffer.GetLength();
      pvInOut = LPTSTR(LPCTSTR(oBuffer));
      pCtxt->WriteClient(pvInOut, &dwSize, 0);

      // Display a page heading.
      oBuffer = "\r\n<CENTER><H2>User Password Check Display</H2></CENTER>";
      dwSize = oBuffer.GetLength();
      pvInOut = LPTSTR(LPCTSTR(oBuffer));
      pCtxt->WriteClient(pvInOut, &dwSize, 0);

      // Display the user's name.
      oBuffer = "\r\n<STRONG>User Name: </STRONG>";
      dwSize = oBuffer.GetLength();
      pvInOut = LPTSTR(LPCTSTR(oBuffer));
      pCtxt->WriteClient(pvInOut, &dwSize, 0);
      dwSize = strlen(pAuthent->pszUser);
      pCtxt->WriteClient(pAuthent->pszUser, &dwSize, 0);

      // Display the user's password.
      oBuffer = "\r\n<P><STRONG>User Password: </STRONG>";
      dwSize = oBuffer.GetLength();
      pvInOut = LPTSTR(LPCTSTR(oBuffer));
      pCtxt->WriteClient(pvInOut, &dwSize, 0);
      dwSize = strlen(pAuthent->pszPassword);
      pCtxt->WriteClient(pAuthent->pszPassword, &dwSize, 0);

      // Create the usual footer.
      oBuffer = "\r\n</BODY>\r\n</HTML>";
      dwSize = oBuffer.GetLength();
      pvInOut = LPTSTR(LPCTSTR(oBuffer));
      pCtxt->WriteClient(pvInOut, &dwSize, 0);

      // return the appropriate status code
      return SF_STATUS_REQ_FINISHED_KEEP_CONN;
   }

   // return the appropriate status code
   return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

This code may look a little complex, or perhaps overly simple, depending on what you were expecting. Let's start breaking it down. The first thing you'll notice in the OnAuthentication( ) function is that we look at the pszUser member of the pAuthent structure. This particular member remains NULL until there is something to verify. However, IIS doesn't fill this structure in for you automatically. Your filter fills the structure in later.

The next thing to do is get the user name from the pCtxt object with the GetServerVariable( ) function. Why do we need to access this object instead of looking at the pAuthent structure? IIS will fill the pCtxt object up for you, but you have to fill in the pAuthent structure.

If there isn't a user name, then we need to ask the user to provide one by displaying a password dialog. Remember that in the past we've used the SF_REQ_SEND_RESPONSE_HEADER request of the ServerSupportFunction( ) call to create a standard header for displaying Web pages from within an ISAPI Filter. In this case, we're going to send an error response to the client using the same technique. Notice, though, that I have to add the error message: "401 Access Denied." You can send any other message that the client will respond to using this same technique (although the server may override some of them when you send them from the OnAuthenticate( ) function).

Along with the 401 error message, a server will also send response headers that list what kind of authentication methods it supports. The two methods that IIS supports right out of the box are NTLM for Windows NT Challenge/Response and Basic for Basic Authentication.

At this point, the client will display a password dialog like the one shown in Figure 9-15. The user will fill in their user name and password, and then click OK. What actually happens is that the user generates an entirely new request. This time the request contains a user name and password. The request will also include the authentication method passed by the server, so that the server will know that the client can support this authentication method. This is the end of the first pass through the filter.

Figure 9-15: The password dialog allows the user to provide a user name and password for the server.

Now we begin the second pass through the security filter. The pszUser member of the pAuthent structure is still NULL. However, the GetServerVariable( ) function does return a user name this time. At this point, we send the client another response header, but it's an OK header this time (the same one we used in previous examples).

We also need to fill in the pAuthent structure. Notice that you have to use the strncopy( ) function to perform this task. We also need to use the GetServerVariable( ) function to retrieve the user password and place it in the pszPassword member of the pAuthent structure. At this point, we've completed the second pass through the security filter.

Let's begin the third, and final, pass through the security filter. This time the pszUser member has a user name in it, so we skip the first section of the function. The second section of the function retrieves the user name and password from the pAuthent structure and uses them to create a Web page. I've talked about this technique in the previous example, so I won't discuss it here again.

Testing the SendErr ISAPI Filter

There are some special considerations for testing the SendErr ISAPI Filter. First, make sure you set the security levels as described in the "Setting Your Security" section of this chapter. Obviously, you'll want to return the security settings to normal once you've finished the SendErr ISAPI Filter test. Like the other filters in this chapter, you'll also need to load the SendErr ISAPI Filter using the procedure outlined in the "Loading the Filter" section of this chapter. Make sure you unload any other filters that you may have loaded on the server because they could affect the results of the test. Figure 9-16 shows the final output of our diagnostic filter.

Figure 9-16: The output of our diagnostic filter allows you to see the results of whatever process the user name and password went through before being passed to the server.

There are two member functions of the pAuthent function that we didn't use in this example. Both the cbUserBuff and cbPasswordBuff members tell you the size of the variables in the structure. Unfortunately, they tell you the maximum size of these variables, which makes them unusable in this example. You need to determine the actual length of the variables using the strlen( ) function for display purposes. However, these member variables will become important when it comes time to get input from the user. For example, you could use them to ensure that the user hasn't entered a password that is either too long or too short.

Modifications Required to Create a Simple Security Filter

It's easy to make the example in Listing 9-6 into a plain security filter. Simply remove the last else clause — the one that creates the Web page. Removing this clause and the code it contains will allow the server to process the security request as normal. You can change the security settings to anything you want to test the modified version of the filter out since we don't need to see a plain-text user name or password. Any security-specific processing you want to perform must occur during the second pass through the filter; before you copy the user's name and password for the server to use.

Working with URL-mapping Request Filters

The URL is the basis for finding just about everything on the Internet. It's no wonder that controlling URLs on your Web site is an important matter. Not only can a URL direct users to information they need, but hackers can also use URLs to help themselves to data you don't want them to see.

CD-ROM Location: Chapt09\ReDir
Program Name: ReDir.DLL
Source Code Modules in Text: ReDir.CPP

URL mapping is a process in which you can redirect a user's request to a more specific, better-controlled or even hidden area of your Web server. The user can enter the URL they want to look at, and an ISAPI Filter can send them to the most appropriate place to look at the requested information. The following list gives you some idea of when URL mapping is especially important.

These are just a few of the potential uses for URL mapping on a Web site. You'll find that URL mapping is one of the more useful things that you can use an ISAPI Filter for. The short take on all of this is that you can use URL mapping as a traffic cop on your Web server, it helps redirect traffic to keep things moving.

Table 9-3
Benefits of URL Mapping

Feature	***Description of Benefit***
Hidden Information	You may want to keep hackers guessing about where you actually store information on your Web server. Virtual directories provide one method of redirecting traffic, but you can use an ISAPI Filter for this purpose as well. The advantage to using an ISAPI Filter is that you can make changes to your mapping configuration with ease, something that takes a lot of time when using virtual directories.
Browser Differences	Every day seems to bring yet another browser compatibility problem to light. The latest problem concerns how various browsers use Java. (Because of certain events that are unfolding right now, the Java compatibility problem may be fixed by the time you read this.) You can take several routes to fix these browser-compatibility problems; one of which is to maintain two separate sets of pages: one for Netscape and another for Microsoft. An ISAPI Filter can detect which browser is knocking at your door and redirect it as needed.
Company-specific Agenda	Imagine, for a second, that you're a user who wants to access the company's Web site with the least amount of effort. Now, imagine you're a manager who wants to use various Web pages for different purposes based on the day of the week. An ISAPI Filter could read the system clock and redirect a user's request to the right Web page; all they would need to do is type in the same URL every day to get to the right Web page on your server. You could make it as simple as typing: `http://www.mycompany.com/today`. For that matter, you could simply make this the user's homepage and they would see the specific Web page for a given day.
Dynamic Reconfiguration	At times, your Web server gets overwhelmed with requests. Wouldn't it be nice if you could detect this condition, then ask another server to help out? ISAPI Filters can do just that. They can help you detect conditions that would normally take your server offline, and redirect those requests to another server with a lighter load.

Continued

Table 9-3 *(continued)*

Feature	***Description of Benefit***
Custom Pages	What if someone is trying to access your Web site and they either don't have the correct software installed or don't belong there in the first place? You can actually create a Web page within your ISAPI Filter and send it to the client. For example, if someone with an old version of Microsoft Internet Explorer visits your Web site, you could tell them where to download a new copy. It doesn't matter which URL they type in; your ISAPI Filter can detect the condition and react to it.
Web-site Statistics	You've probably seen more site counters than you care to remember on the Internet. However, most of these counters have the same problem: they can actually miscount the number of people visiting a site. In addition, you really don't know if the person has visited a particular area of your Web site using this method. URL mapping may not involve moving a user from one location to another in some cases. You may simply want to monitor their movement between various areas of your site and then record those movements for later use. This is especially important when you want to create a virtual directory setup when the normal logging methods may not work.

I've used filters that send output to the client as a diagnostic aid in more than one situation. The most important thing that a diagnostic filter can do is show you the actual flow of data between the client and the server. A well-designed diagnostic filter can help you understand why a filter or extension isn't working as anticipated. That's one of the reasons I chose a test filter in this chapter—to show you the power of using the WriteClient( ) function as a diagnostic or informational aid.

The most common way to use an URL mapping filter is to switch one location on your Web server with another. All you need to do is change the physical location information. This section of the chapter will look at the simplest example of URL mapping—exchanging one URL for another. We'll look for a particular URL, then map it somewhere else on the server. Obviously, you can use this technique to detect a wide variety of conditions. The most common use would be to mimic the effects of using virtual directories to thwart the efforts of hackers who visit your site. However, you can also use this technique to differentiate between a novice and an advanced user (the advanced user might get mapped to a site that contains more options and allows better access to the Web site with less work). Looking for a particular URL is only one way to make URL mapping an extremely important part of your ISAPI toolkit.

Quite a bit of information is available about URL mapping on Microsoft sites on the Internet. The OnUrlMap( ) function documentation will probably continue to change as Microsoft enhances ISAPI as a whole. You can get updated information about using the OnUrlMap( ) function at: `http://premium.microsoft.com/msdn/library/devprods/vc++/vcmfc/d3/s402b2.htm`. (This is just one of several places to find OnUrlMap( ) documentation online.)

Another good place to look for an overview of ISAPI Filters and how URL mapping fits in is at: `http://premium.microsoft.com/msdn/library/techart/isfilter.htm`. This particular site is a work in progress and includes an example of how to create a filter to convert Word, Excel, and text documents to HTML.

If you want a general list of ISAPI Filter examples, then take a look here: `http://premium.microsoft.com/msdn/library/devprods/vc++/progsguide/f26/d29/s4cdb8.htm`. You'll find at least one example on URL mapping along with others that demonstrate the other types of filters described in this book.

Creating the ReDir ISAPI Filter Shell

The first step is to create the ReDir ISAPI Filter shell. We'll use the same basic technique that I've shown earlier in this chapter. The following steps will talk about some differences from that procedure that we'll need for this particular example.

1. Create a basic ISAPI Filter using the ISAPI Extension Wizard. I gave mine the name of ReDir, but you can use any name you like.
2. Check the ISAPI Filter option and uncheck the ISAPI Extension option on the first page of the MFC Extension Wizard.
3. Chose to monitor both secured and nonsecured port sessions.
4. Check the URL mapping requests option and uncheck the End of connection option. Figure 9-17 shows the New Project Information dialog for this project.

New Project Information
ISAPI Extension Wizard will create a new skeleton project with the following specifications:
Internet Server Extension DLL project ReDir.mak
DLL name: ReDir.dll
DLL code in ReDir.h and ReDir.cpp
Resources in ReDir.rc
This project will have a server filter class named CReDirFilter.
+ Your filter will be notified at a low priority.
+ Your filter will be notified about nonsecured and secured connections.
+ Your filter will have handlers for these notification(s):
+ URL mapping requests
Your project will be configured to use MFC in a DLL.
Project Directory:
D:\WIN_NT\CHAPT09\ReDir
OK
Cancel

Figure 9-17: The New Project Information dialog for the ReDir ISAPI Filter

Adding Code to the ReDir ISAPI Filter

Now that we have an application shell, it's time to add some code. Remember that we'll look for a specific page request from the client, then remap it to another location on the Web site. Listing 9-7 shows the OnUrlMap( ) method code for this filter.

Listing 9-7: **OnUrlMap( ) Method Source Code**

```
DWORD CReDirFilter::OnUrlMap(CHttpFilterContext* pCtxt,
    PHTTP_FILTER_URL_MAP pMapInfo)
{
    // Check for the proper URL request.
    if (strnicmp(pMapInfo->pszURL, "/default.htm", 12) == 0)

        // Remap to a redirected Web page.
        strncpy(pMapInfo->pszPhysicalPath,
            "C:\\Inetpub\\wwwroot\\REDIR.HTM",
            pMapInfo->cbPathBuff - 1);

    // return the appropriate status code
    return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

The code is relatively simple: All you need to do is compare the URL in the pszURL member of the PHTTP_FILTER_URL_MAP structure to the one that you're looking for. In this case, we're looking for requests for the main Default.HTM page. Notice that I use the strnicmp( ) function for comparison purposes since we can't directly compare the two strings (the strncmp( ) function would allow you to make a case-sensitive comparison, should you wish to do so).

Once we've determined that the user has requested a filtered Web site, we can use the strncpy( ) function to change the physical location variable, pszPhysicalPath. It's important to use strncopy( ) instead of strcpy( ) so that you don't accidentally overrun the pszPhysicalPath variable. If we had wanted to change the user's browser URL as well as remap the location, we'd need to change the pszURL variable contents as well. In most cases, this really isn't necessary because the redirection will take place every time the user requests the specified page. It really shouldn't be important to the user where they get redirected, as long as they get the information they are looking for.

Designing a Test Web Page

Now that we have a filter, we'll have to create a redirection Web page as well. In this case, I used the name REDIR.HTM in the filter source, so that's the name we'll use for the Web-page file as well. Listing 9-8 contains the relatively simple source code for the Web page in this example.

Listing 9-8: **REDIR.HTM Source Code**

```
<HTML>
<HEAD>
<TITLE>Redirection Page</TITLE>
</HEAD>
<BODY>

<!- Display a heading. ->
<CENTER><H2>Redirection Center</H2></CENTER>

<!- Display a message. ->
You were redirected here from somewhere else.<P>
Click <A HREF="DEFAULT.HTM">here</A> to see the default page (if you can).

</BODY>
</HTML>
```

All that this code does is display a simple Web page with one link heading back toward our default Web page. Figure 9-18 shows the output of this Web page.

Redirection Page - Microsoft Internet Explorer
File Edit View Go Favorites Help
Search Favorites History
Address http://nt_server/default.htm
Links
Redirection Center
You were redirected here from somewhere else.
Click here to see the default page (if you can).
Local intranet zone

Figure 9-18: Our simple redirection Web page is what you'll see if you try to display the default Web page while the ReDir1 ISAPI Filter is running.

If you try clicking the link shown on the redirection Web page, you'll find that you can't get to the default Web page as long as the ReDir1 ISAPI Filter is installed. Notice, though, that the Address field on the Internet Explorer browser still shows that we're at the default.htm page. Using URL mapping won't change what the user perceives as the correct Web site. Only you will know that they're looking at another location on your sever.

Summary

ISAPI Filters do exactly what the name implies; they filter the data flowing between the client and server. Filtering can take a lot of different forms, as demonstrated in this chapter. While the filtering capabilities that ISAPI provides are powerful, there are often easier-to-implement solutions that you should normally consider trying first. Use an ISAPI Filter when these other solutions won't work. Here are some additional conclusions you can draw from the contents of this chapter:

- You can create seven different types of ISAPI Filters.
- Each ISAPI Filter type monitors a different server event, which corresponds to a particular phase of client-request processing.
- Most of the ISAPI Filter event handlers are provided with a pointer to a data structure specifically designed to handle that event.
- In some cases, the data structure pointer provided to an ISAPI Filter event handler includes one or more pointers to server functions.
- ISAPI Extensions normally deal with direct user requests and always output a Web page.
- ISAPI Filters normally work in the background and seldom create Web-page output, although it's certainly possible to do so.
- Request headers can contain information about the user and the request they're making, and three common elements are always provided: request method, URL, and HTTP version.

✦ ✦ ✦